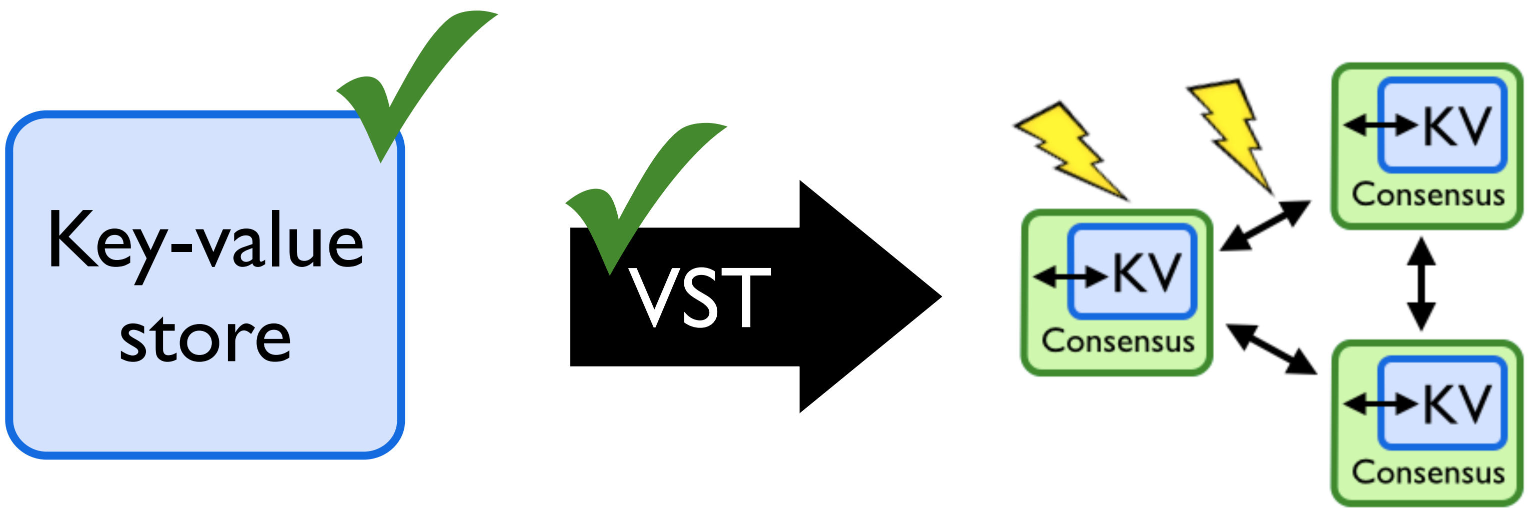
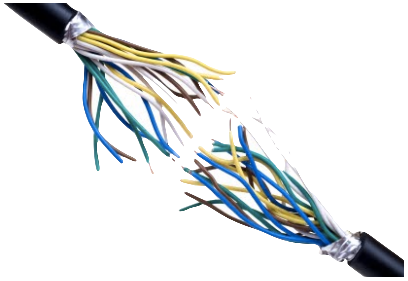
Verdi: A Framework for Implementing and Formally Verifying Distributed Systems



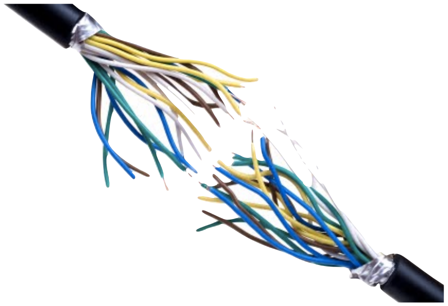
James R. Wilcox, Doug Woos, Pavel Panchekha,
Zach Tatlock, Xi Wang, Michael D. Ernst, Thomas Anderson

Challenges

Distributed systems run in unreliable environments



Many types of failure can occur



Fault-tolerance mechanisms are challenging to implement correctly

Challenges

Distributed systems run in unreliable environments

Many types of failure can occur

Fault-tolerance mechanisms are challenging to implement correctly

Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

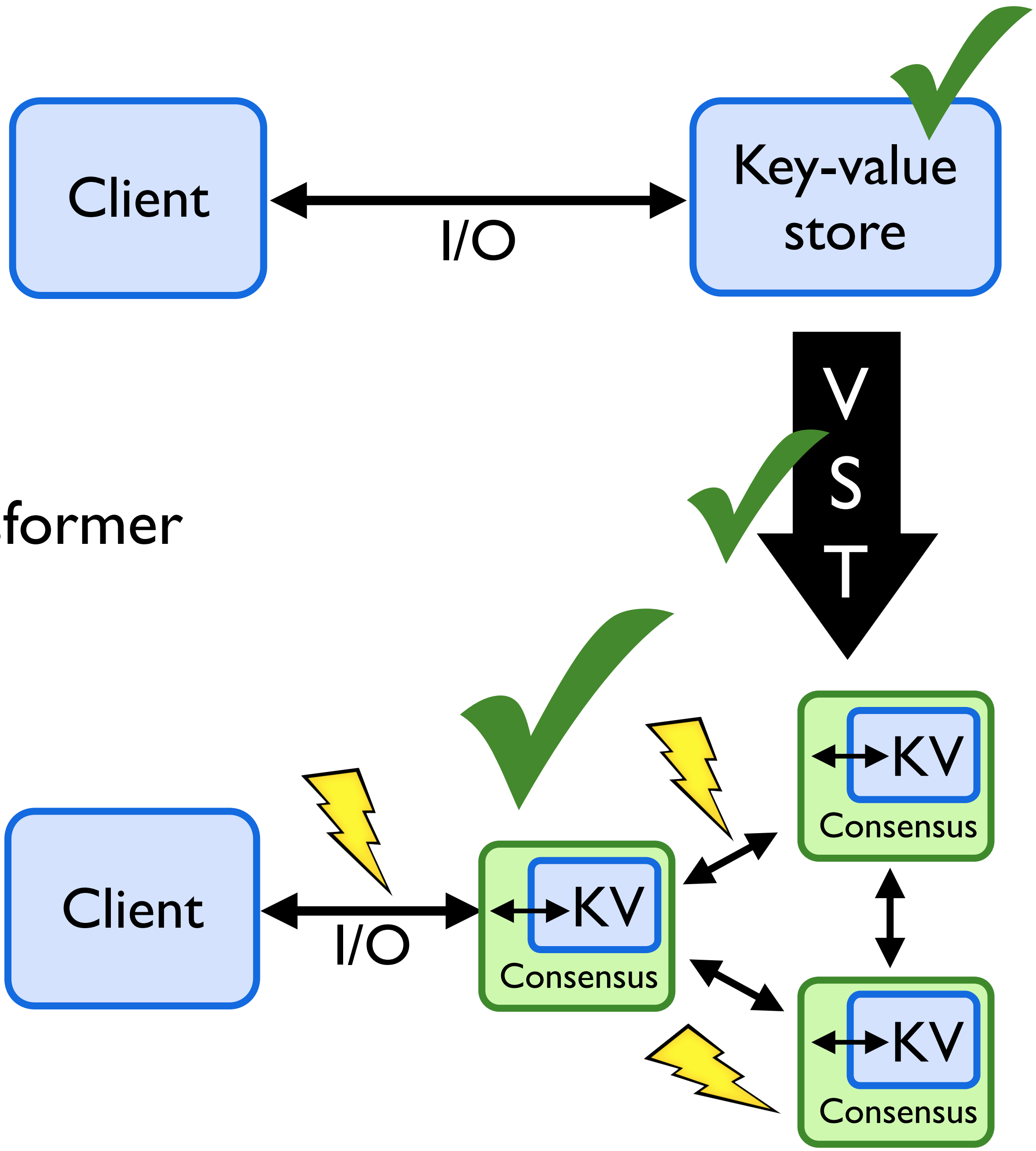
Verify fault-tolerance as transformation between semantics

Verdi Workflow

Build, verify system
in simple semantics

Apply verified system transformer

End-to-end correctness
by composition



Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics

General Approach

Find environments in your problem domain

Formalize these environments as operational semantics

Verify layers as transformations between semantics

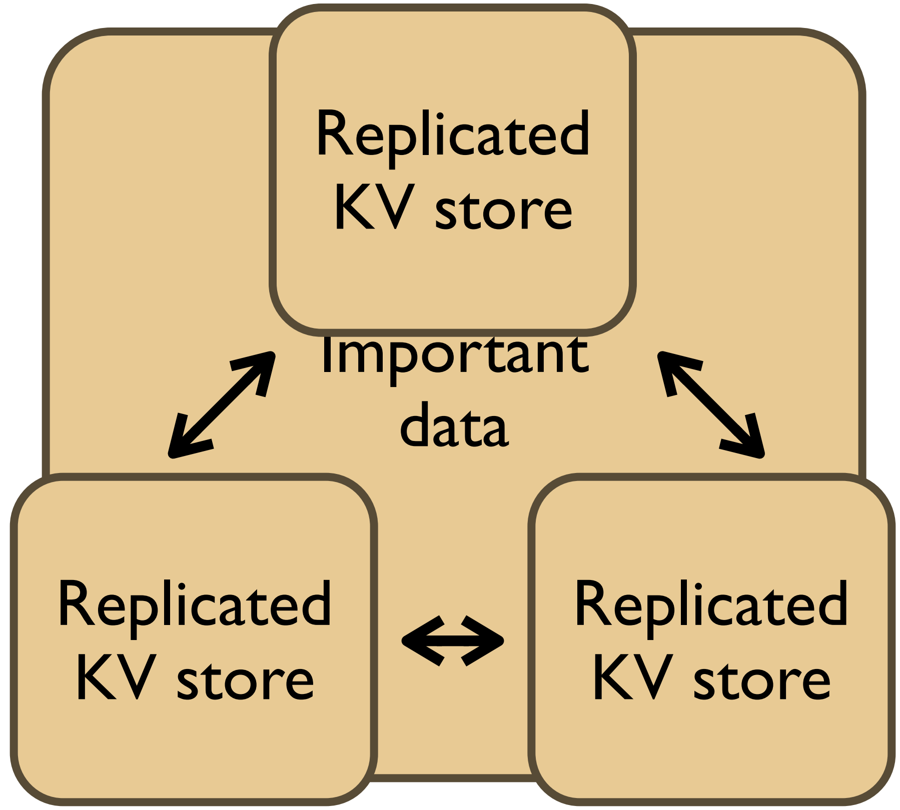
Verdi Successes

Applications

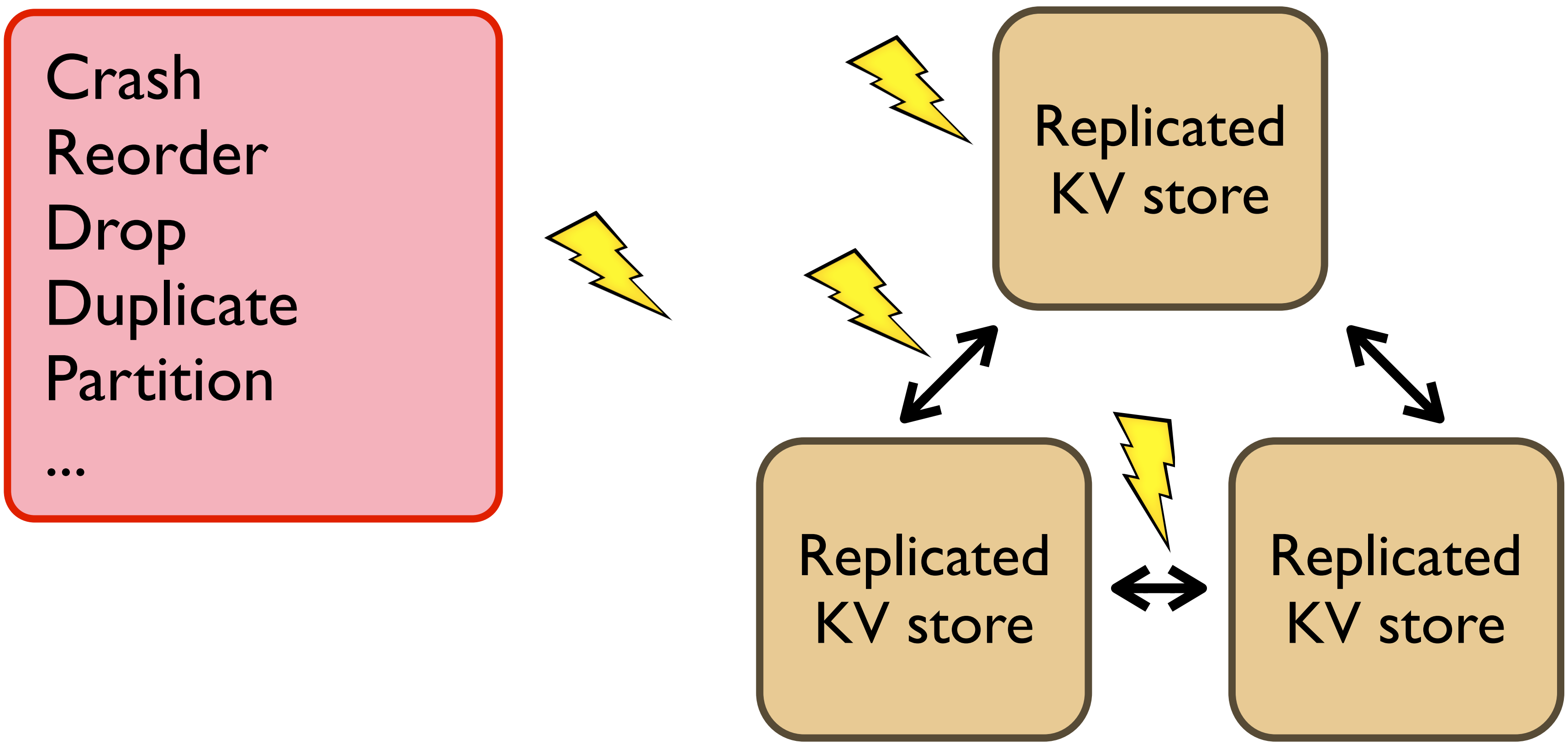
- ☆ *Key-value store*
- *Lock service*

Fault-tolerance mechanisms

- *Sequence numbering*
- *Retransmission*
- *Primary-backup replication*
- ☆ *Consensus-based replication linearizability*

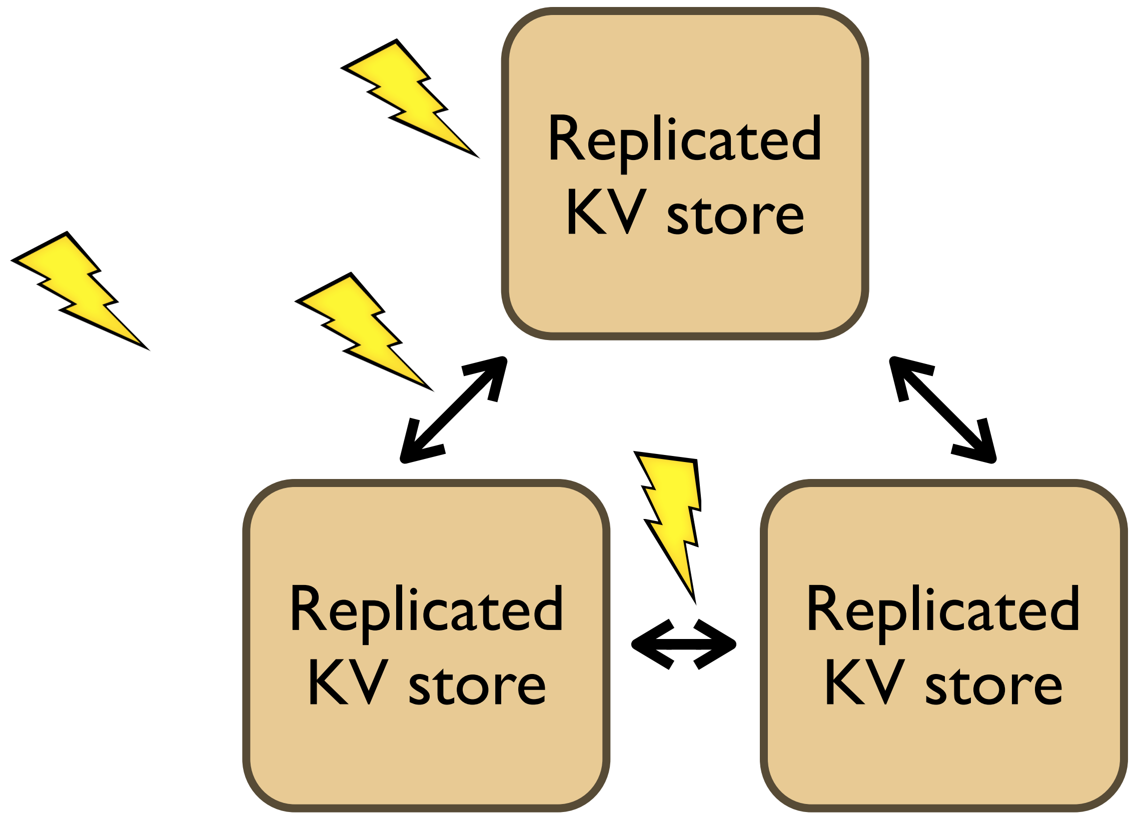


Replicated for availability



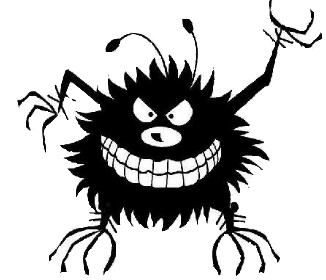
Environment is unreliable

Crash
Reorder
Drop
Duplicate
Partition
...



Decades of research; still difficult to implement correctly

Implementations often have bugs

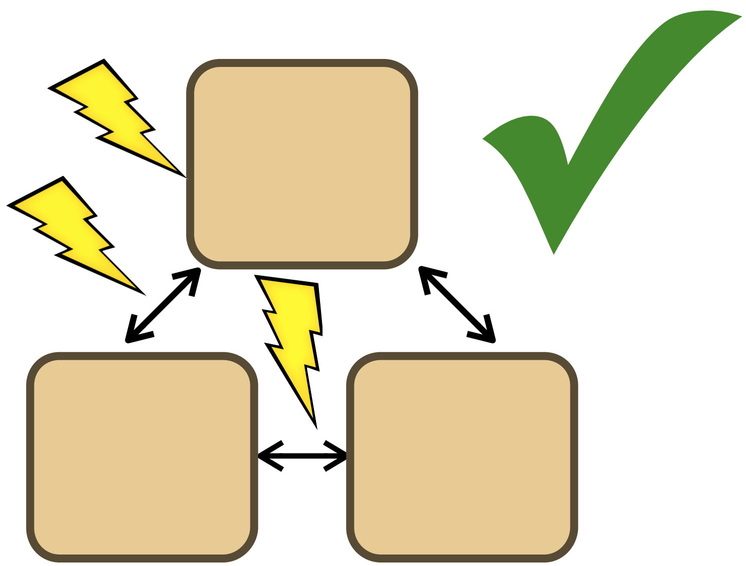


Bug-free Implementations

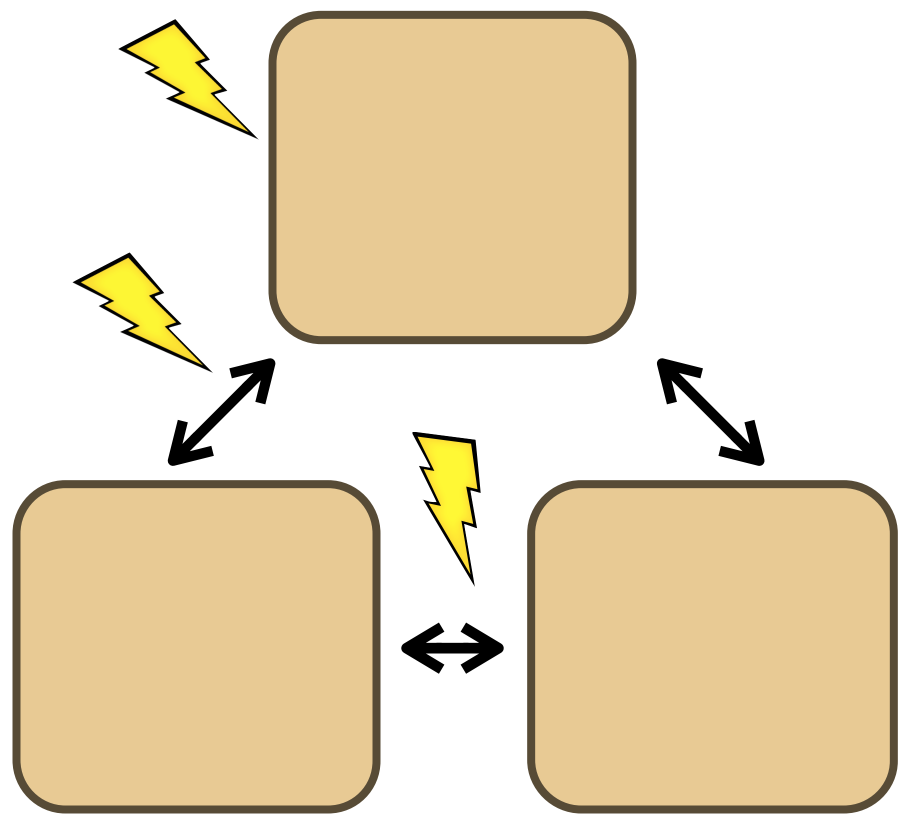
Several inspiring successes in formal verification

CompCert, seL4, Jitk, Bedrock, IronClad, Frenetic, Quark

Goal: formally verify distributed system implementations

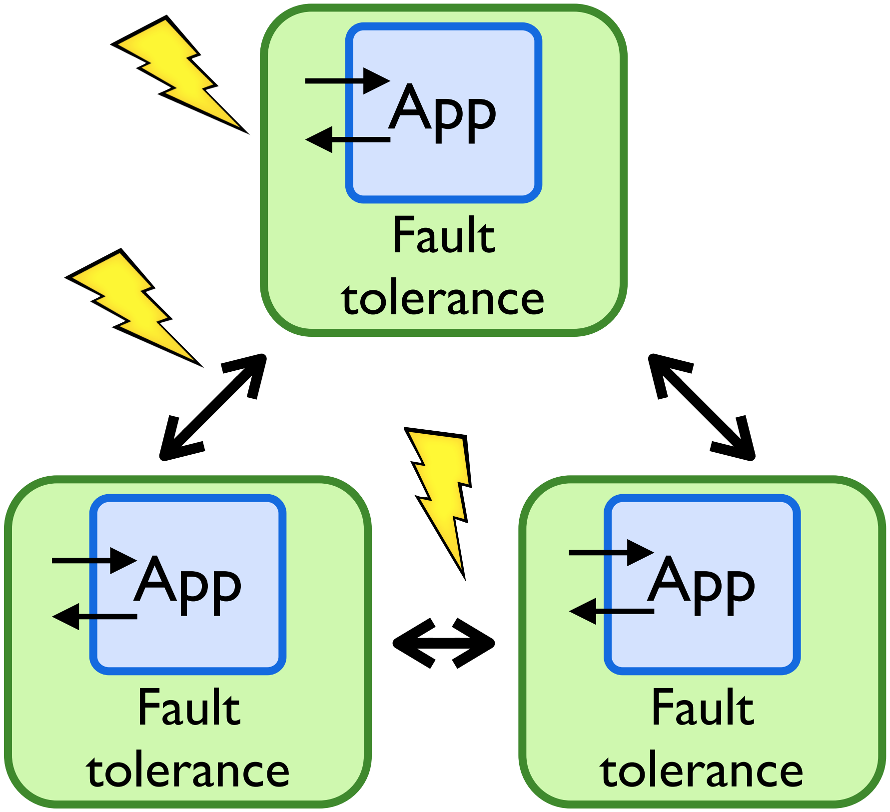


Formally Verify Distributed Implementations



Separate independent system components

Formally Verify Distributed Implementations



Separate independent system components

Verify application logic independently from fault tolerance

Formally Verify Distributed Implementations

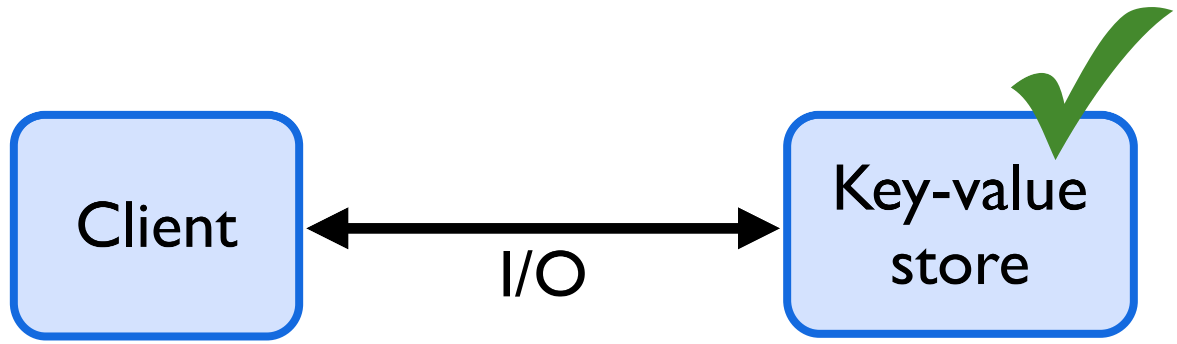
1. Verify application logic
2. Verify fault tolerance mechanism
3. Run the system!

Separate independent system components

Verify key-value store independently from consensus

1. Verify Application Logic

Simple model,
prove "good map"

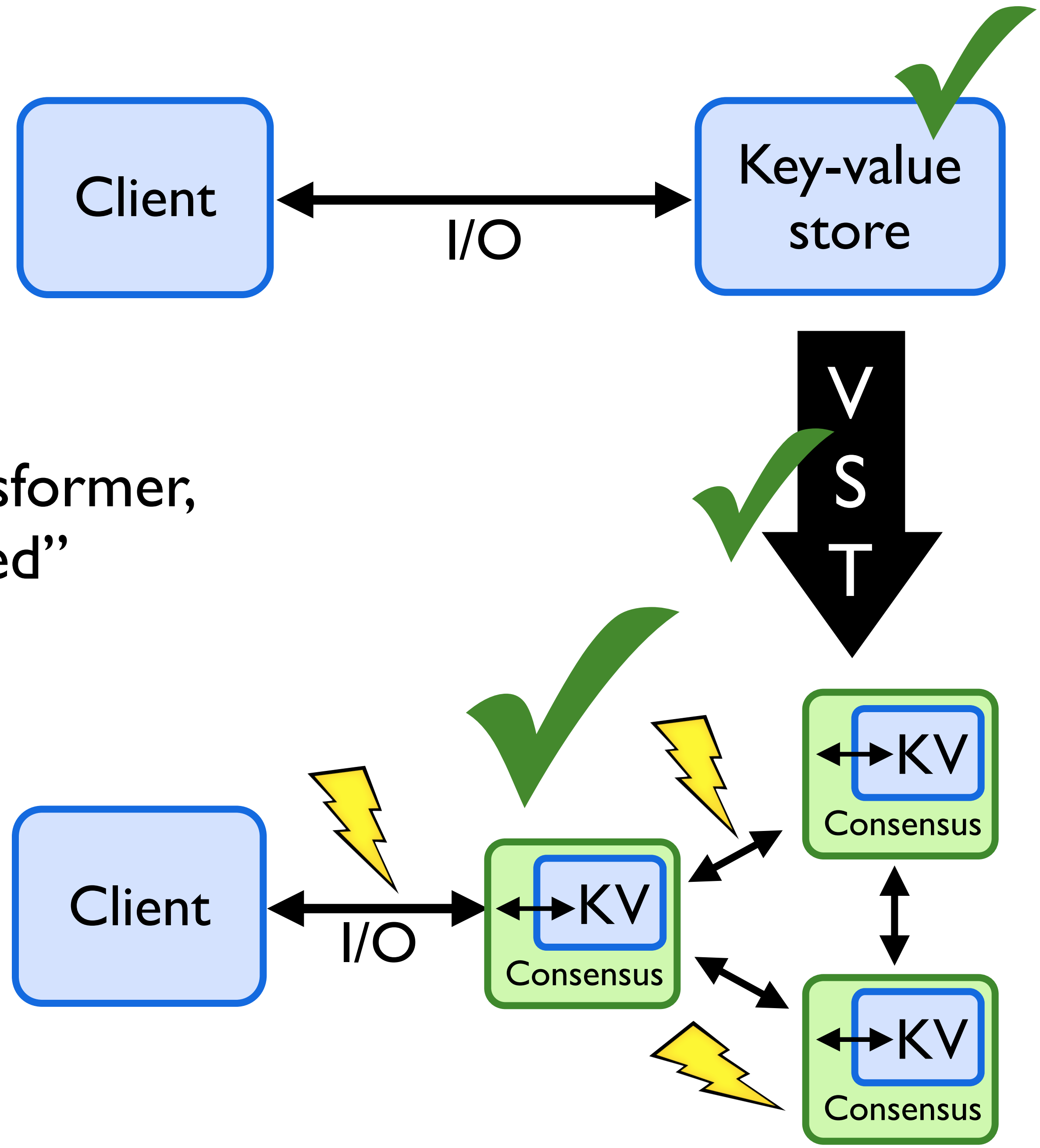


2. Verify Fault Tolerance Mechanism

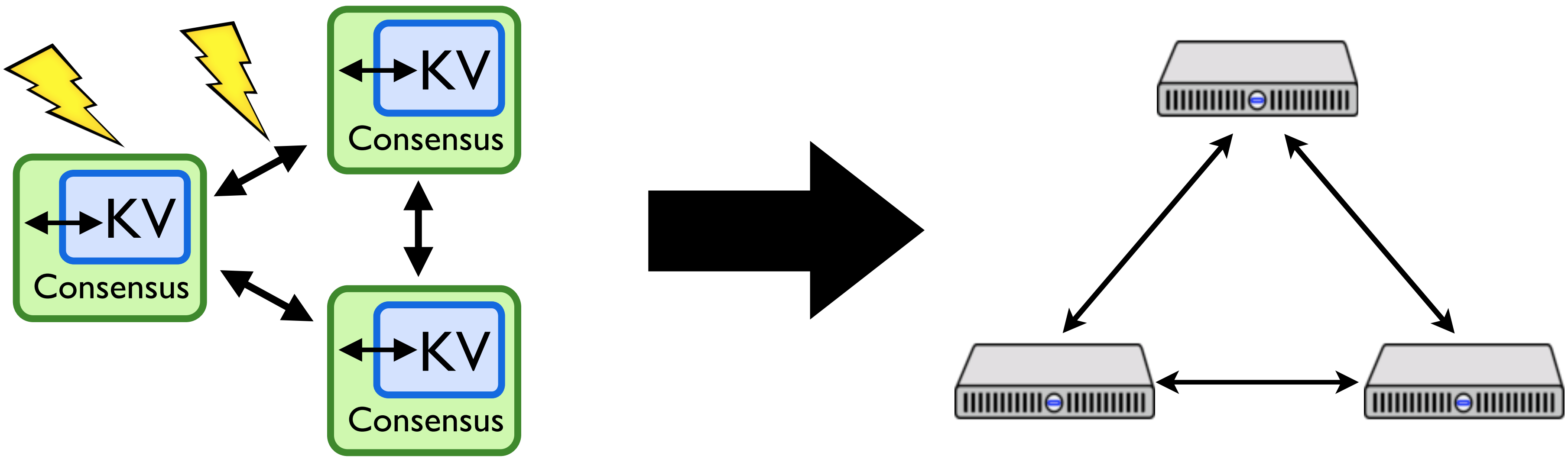
Simple model,
prove "good map"

Apply verified system transformer,
prove "properties preserved"

End-to-end correctness
by composition

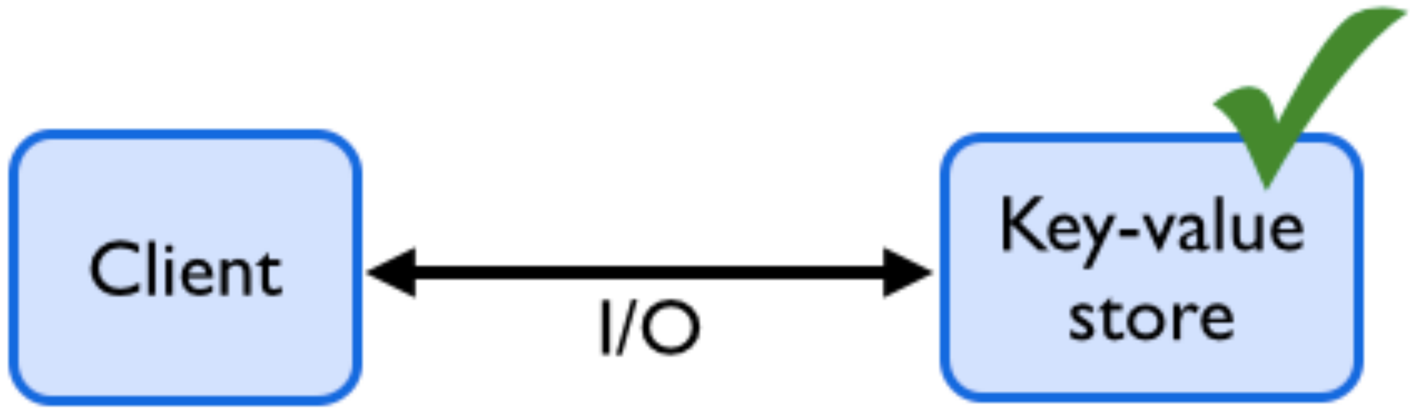


3. Run the System!



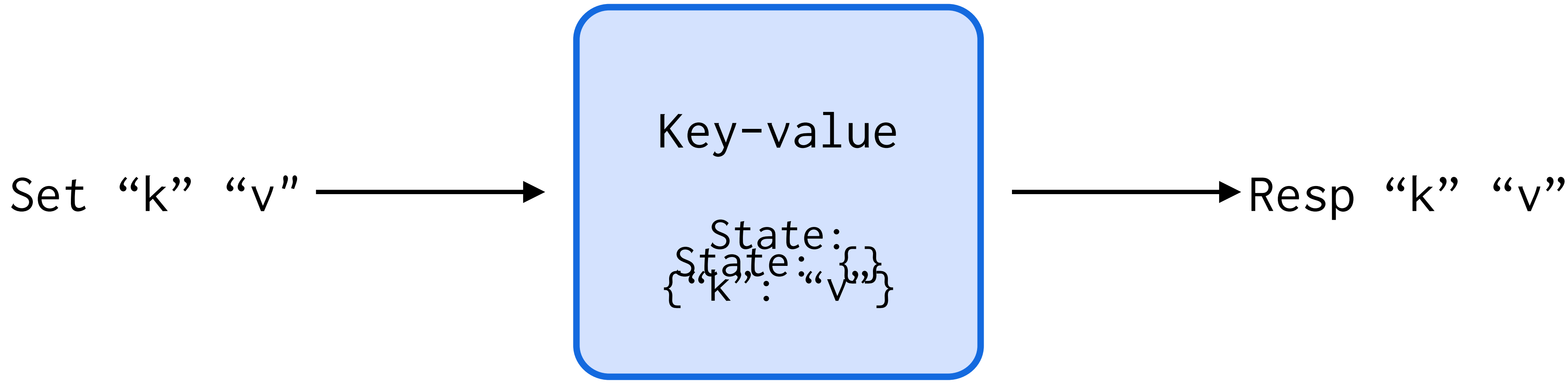
Extract to OCaml, link unverified shim

Run on real networks



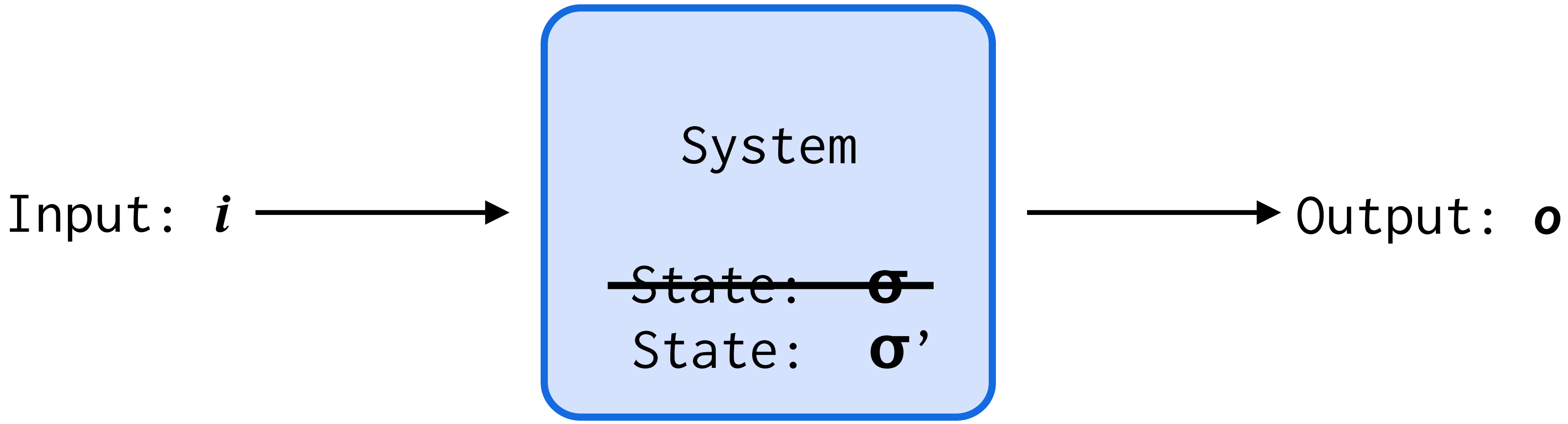
Verifying application logic

Simple One-node Model



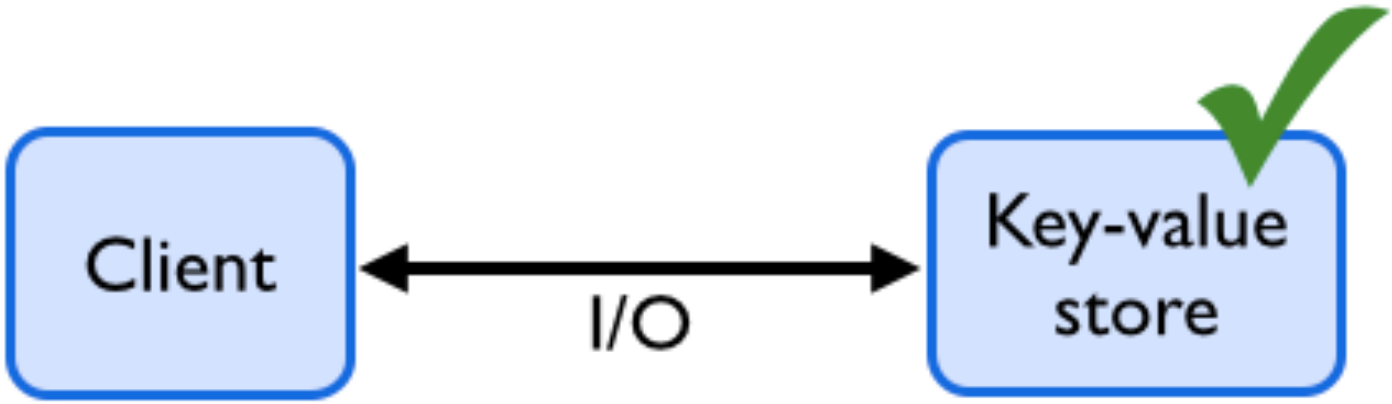
Trace: [Set “k” “v”, Resp “k” “v”]

Simple One-node Model



Trace: [i , o]

Simple One-node Model



Spec: operations have expected behavior (good map)

- Set, Get
- Del, Get

Verify system against semantics by induction

- Safety Property

Verifying Fault Tolerance

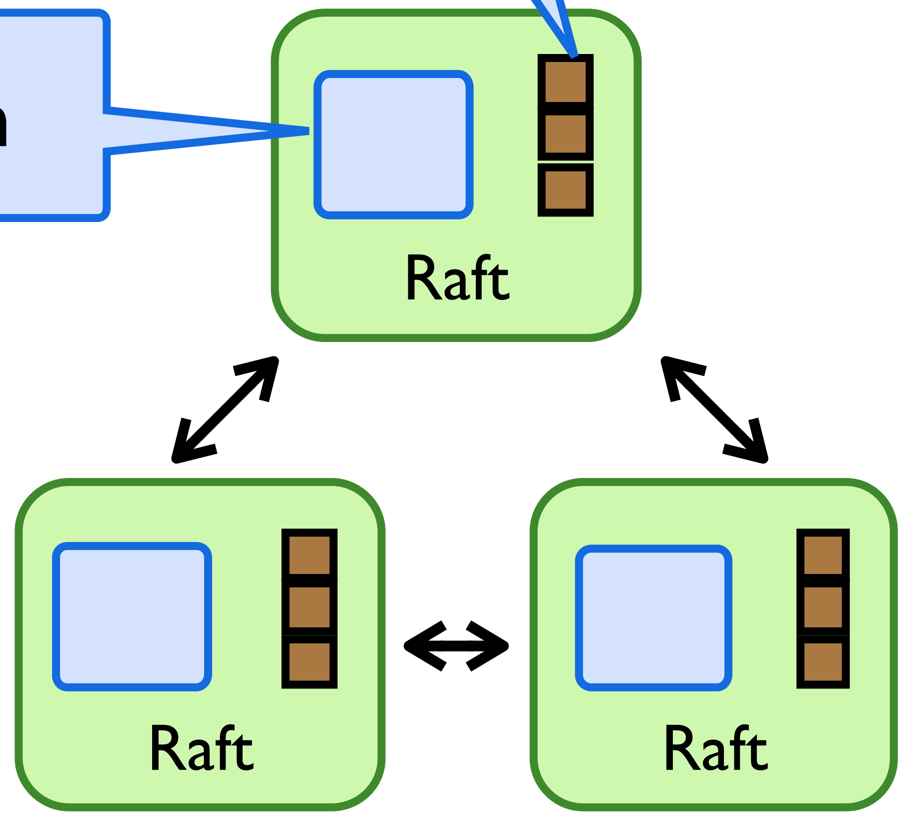
The Raft Transfor

Consensus

replicated

Same inputs on each node

Calls into original system



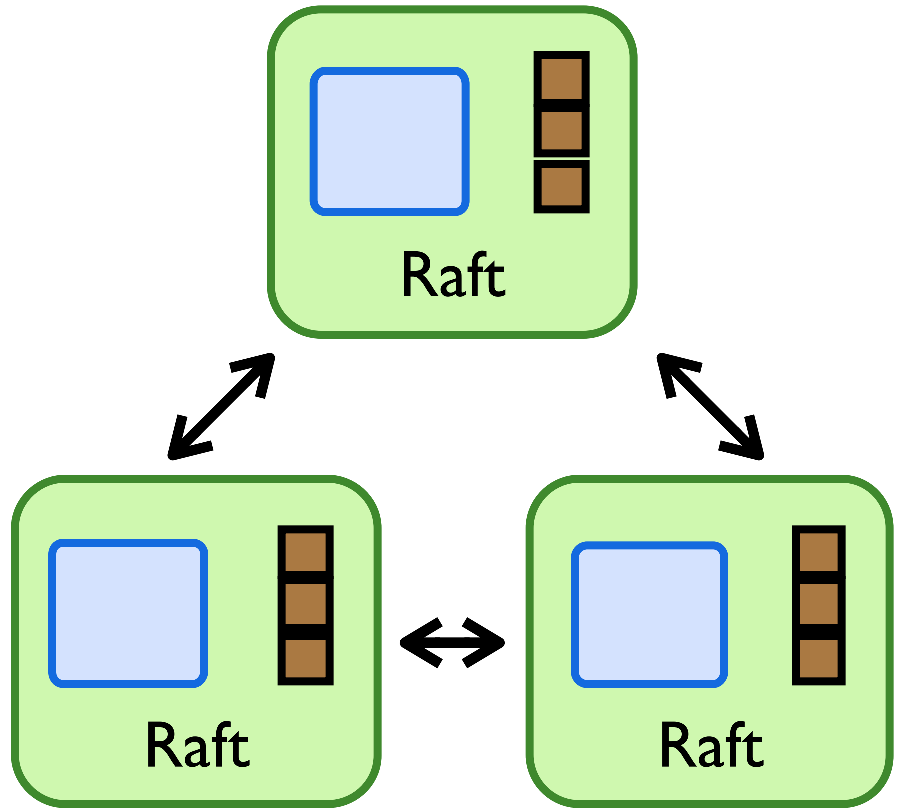
The Raft Transformer

When input received:

- Add to log
- Send to other nodes

When op replicated:

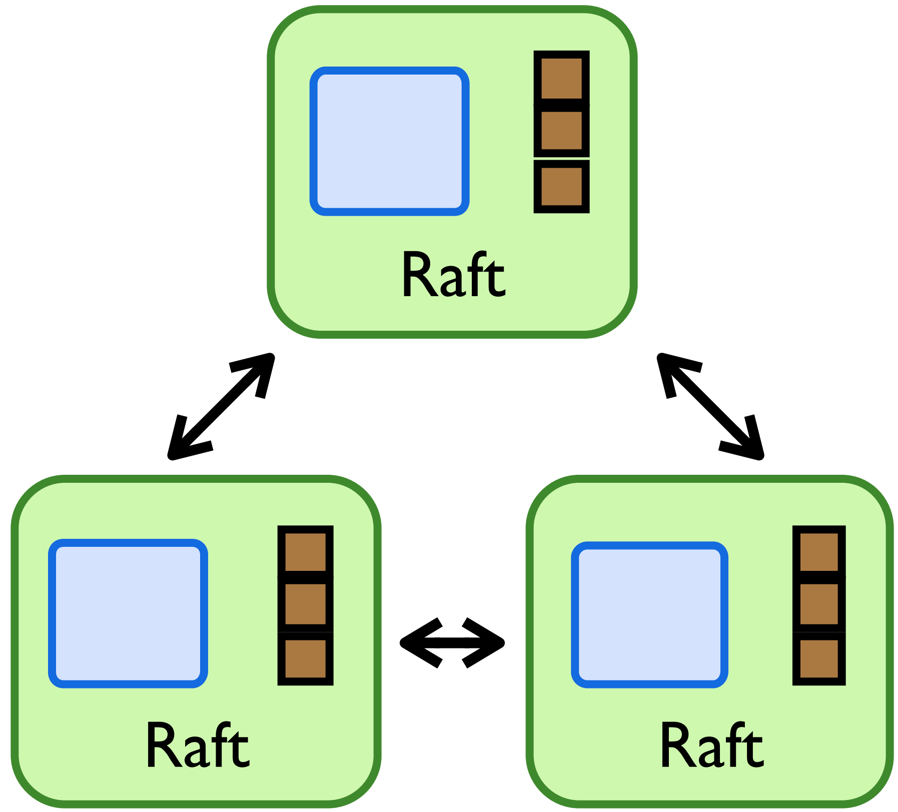
- Apply to state machine
- Send output



The Raft Transformer

For KV store:

- Ops are Get, Set, Del
- State is dictionary

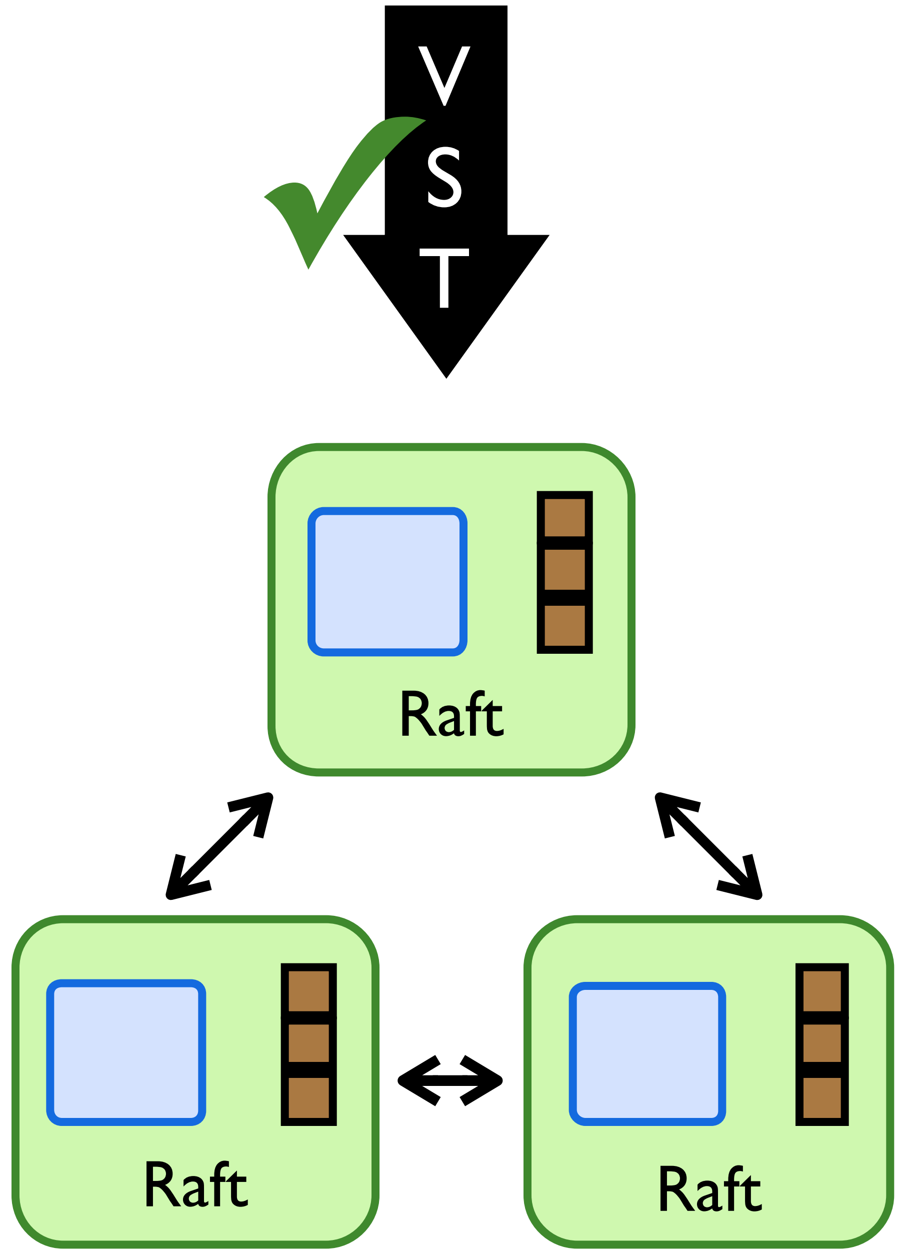


Raft Correctness

Correctly transforms systems

Preserves traces

Linearizability

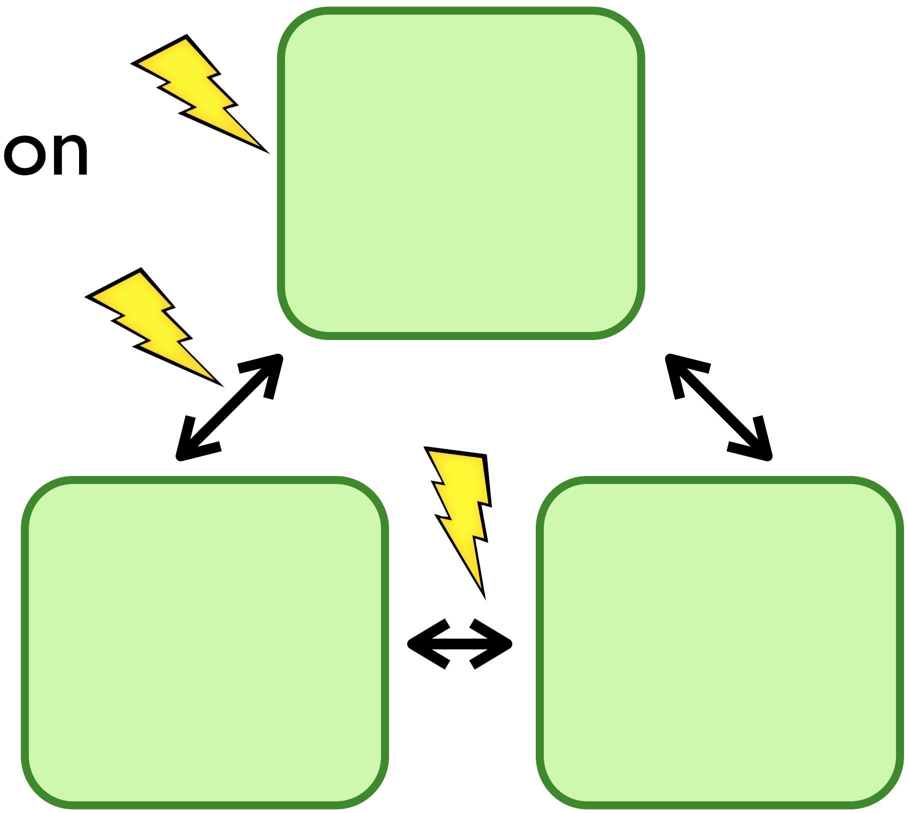


Fault Model

Model global state

Model internal communication

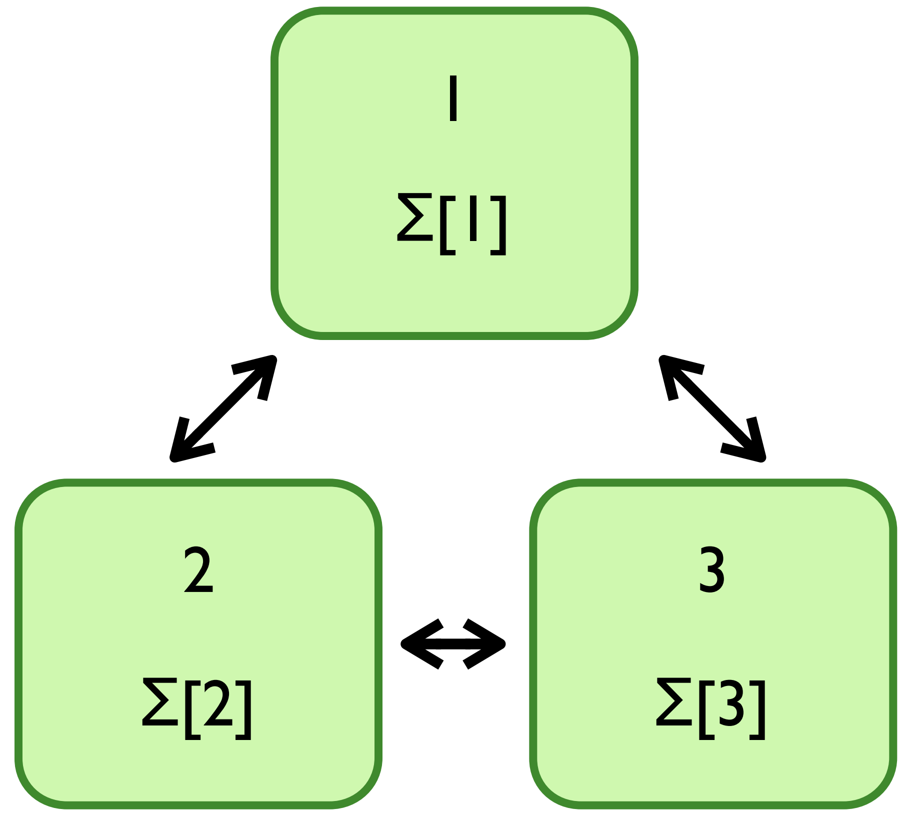
Model failure



Fault Model: Global State

Machines have names

Σ maps name to state

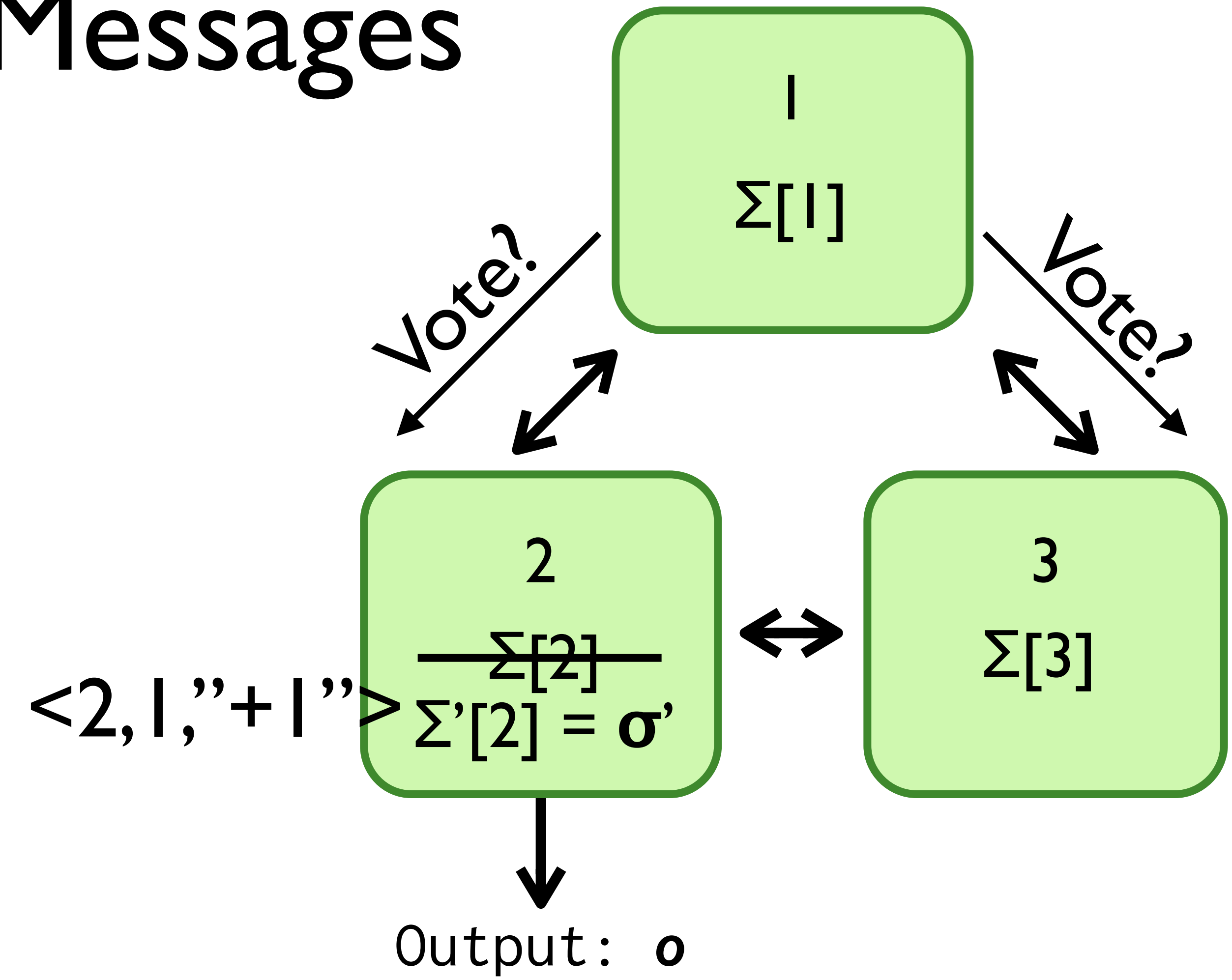


Fault Model: Messages

Network

<1,2,”Vote?”>

<1,3,”Vote?”>

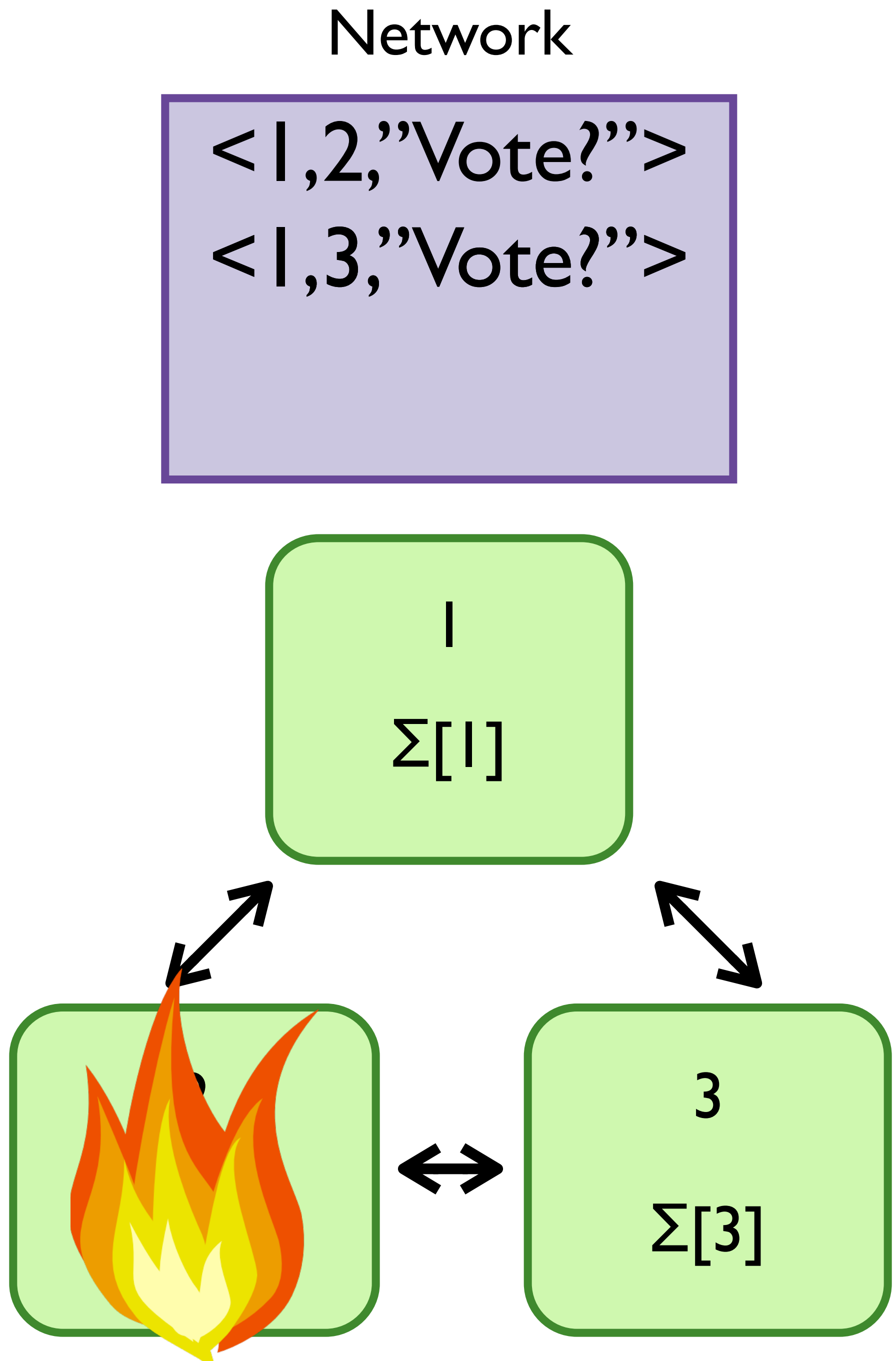


Fault Model: Failures

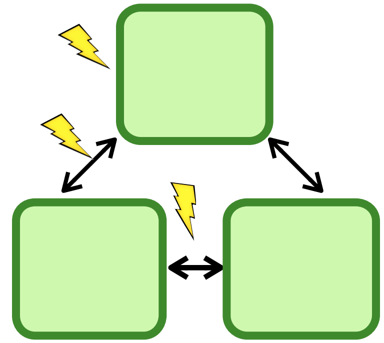
Message drop

Message duplication

Machine crash



Fault Model: Drop



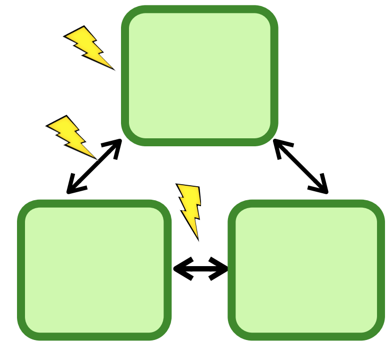
Network

<1,2,"hi">

<1,3,"hi">

$$\frac{}{(\{p\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P,\ \Sigma,\ T)}$$

Toward Verifying Raft



General theory of linearizability

1k lines of implementation, 5k lines for linearizability

State machine safety: 30k lines

Most state invariants proved, some left to do

Verified System Transformers

Functions on systems

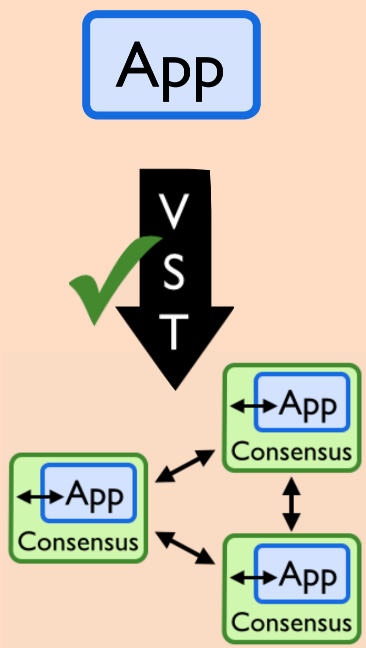
Transform systems between semantics

Maintain equivalent traces

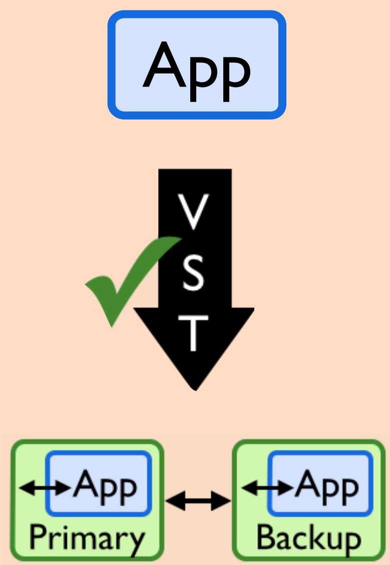
Get correctness of transformed system for free

Verified System Transformers

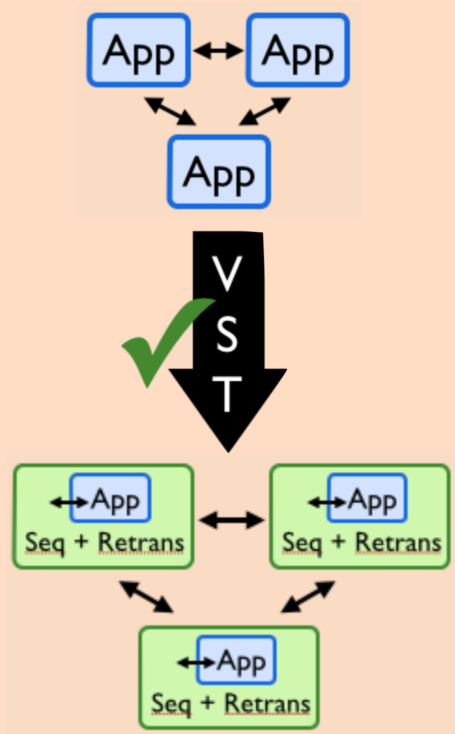
Raft Consensus



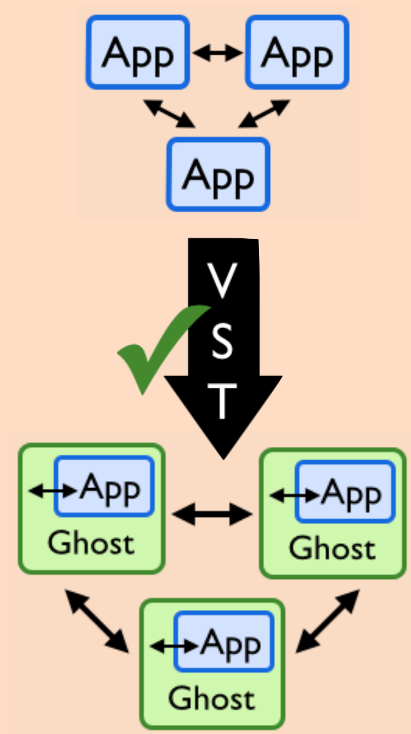
Primary Backup

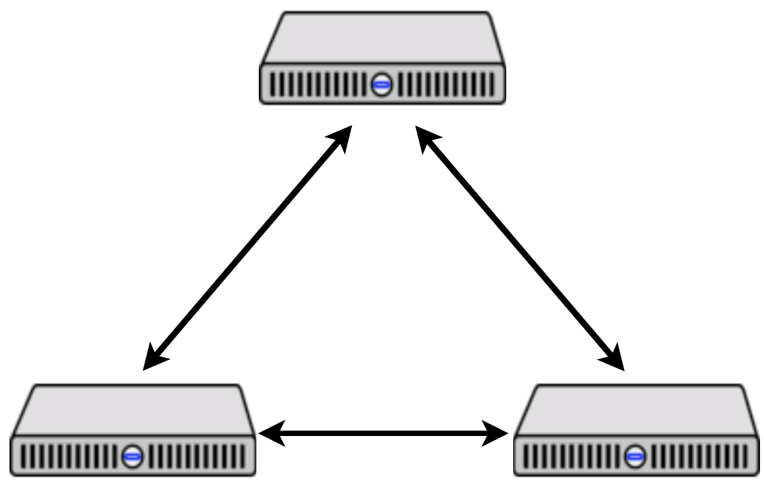


Seq # and Retrans



Ghost Variables

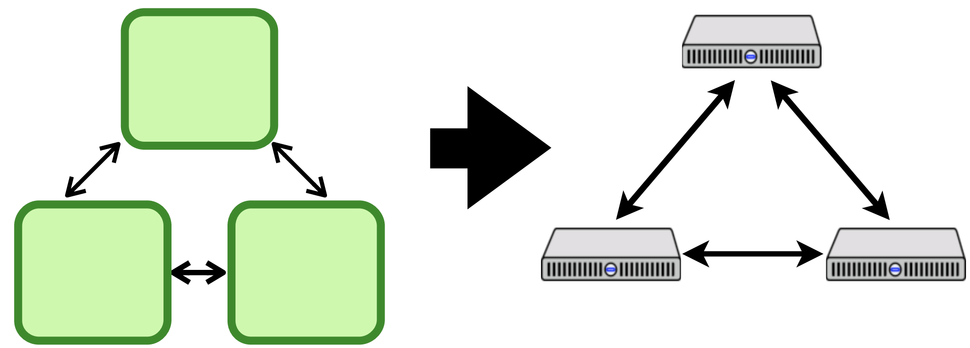




Running Verdi Programs

Running Verdi Programs

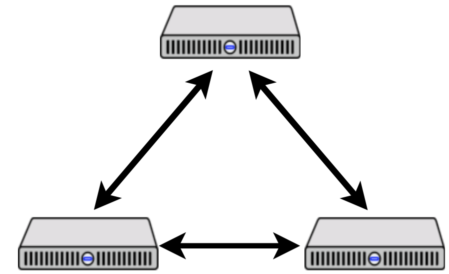
Coq extraction to Ocaml



Thin, unverified shim

Trusted compute base: shim, Coq, Ocaml, OS

Performance Evaluation



Compare with etcd, a similar open-source store

10% performance overhead

Mostly disk/network bound

etcd has had linearizability bugs

Previous Approaches

EventML [Schiper 2014]

Verified Paxos using the NuPRL proof assistant

MACE [Killian 2007]

Model checking distributed systems in C++

TLA+ [Lamport 2002]

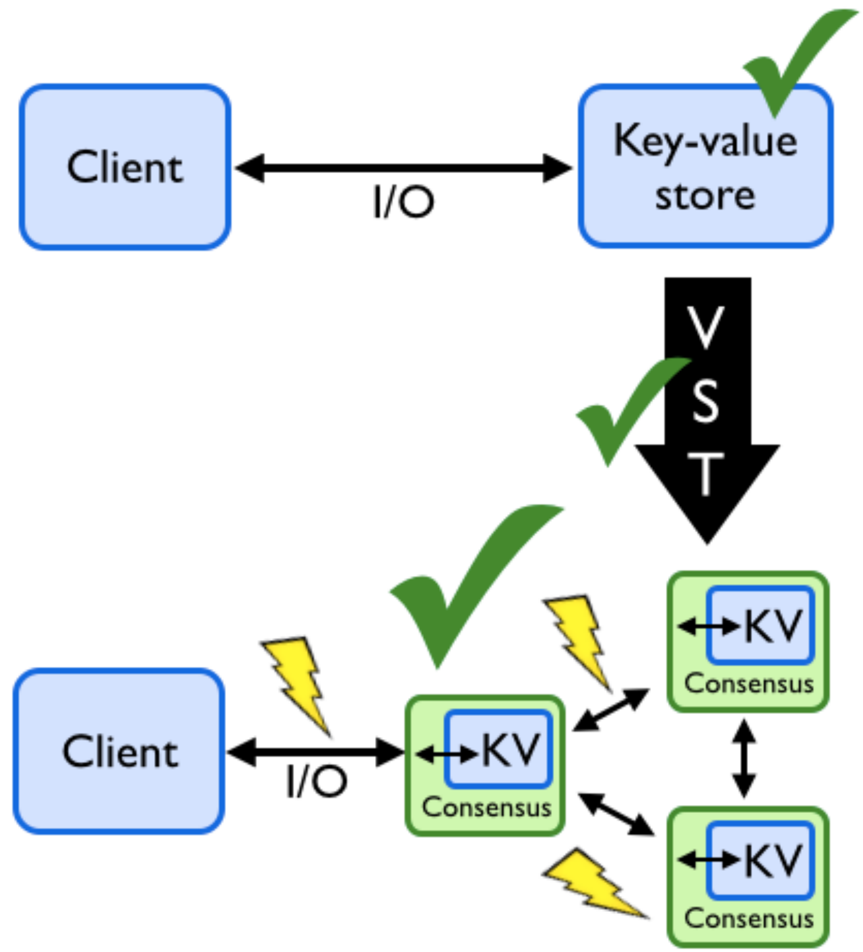
Specification language and logic

Contributions

Formalize network as operational semantics

Build semantics for a variety of fault models

Verify fault-tolerance as transformation between semantics



Thanks!

http://verdi.uwplse.org